Core Java
Multithreading

Akshita Chanchlani

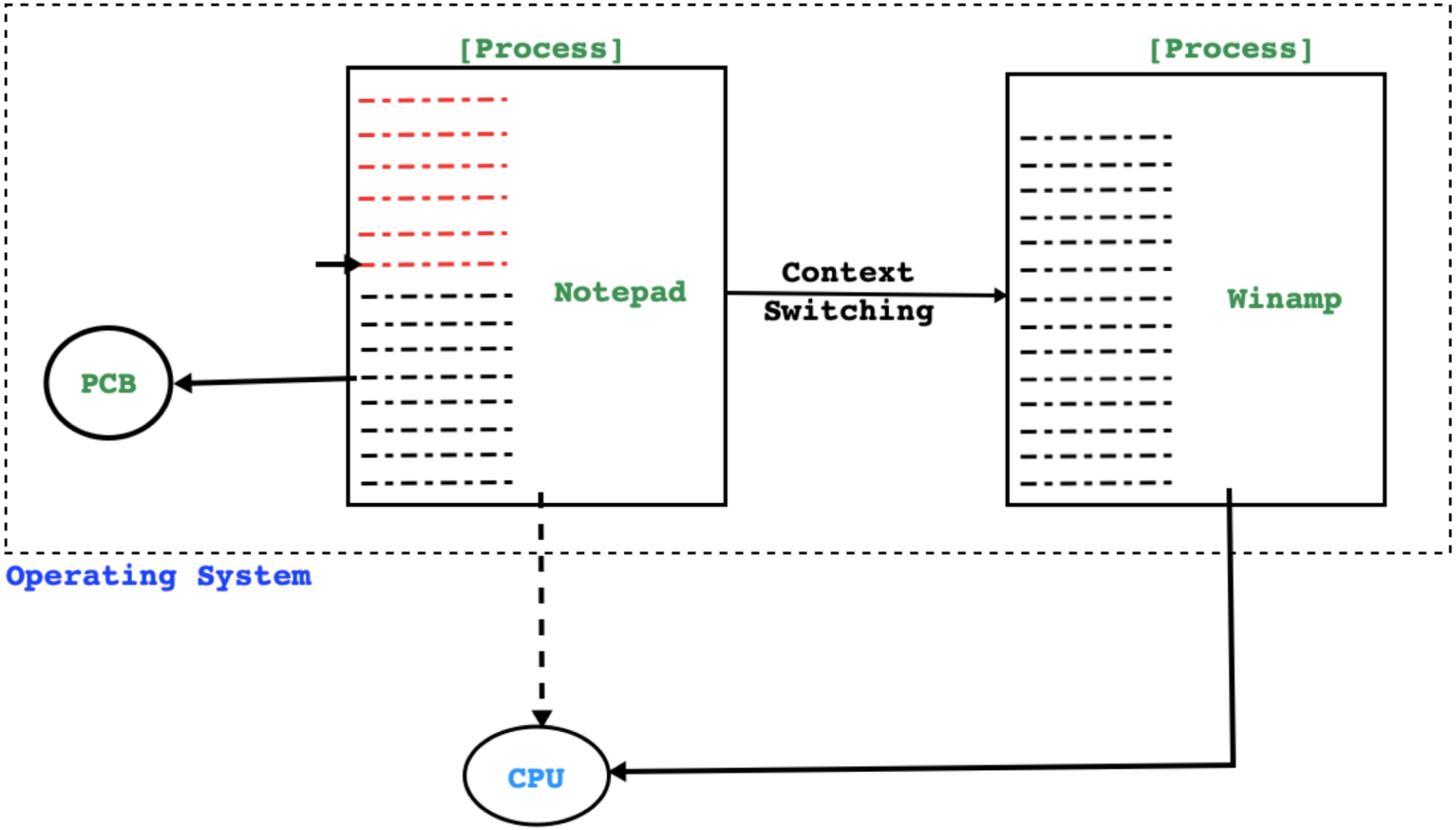
Process Introduction

- Process(also called as task)
  - Program in execution is called as process.
  - Running instance of a program is also called as process.
- Ability of an operating system to execute single process at a time is called as single tasking operating system.
  - Example : Microsoft Disk Operating System(MSDOS)
- Ability of an operating system to execute multiple processes at a time is called as multi tasking operating system
  - Example : Microsoft Windows, Unix/Linux, Mac OS
- This ability to run multiple processes or application at a time is called as concurrency in oops.

Thread Introduction

- Thread(also called as task)
  - Light weight process or sub process is also called thread.
  - In the context of Java, thread is a separate path of execution which runs independently.
- Thread is a operating system / Non Java resource.
- If we want to utilize hardware resources(e.g. CPU) efficiently then we should use thread.
- If application take help of single thread to execute application then it is called single threaded application.
- If application take help of multiple threads to execute application then it is called multi threaded application.
- Every Java application is multithread.

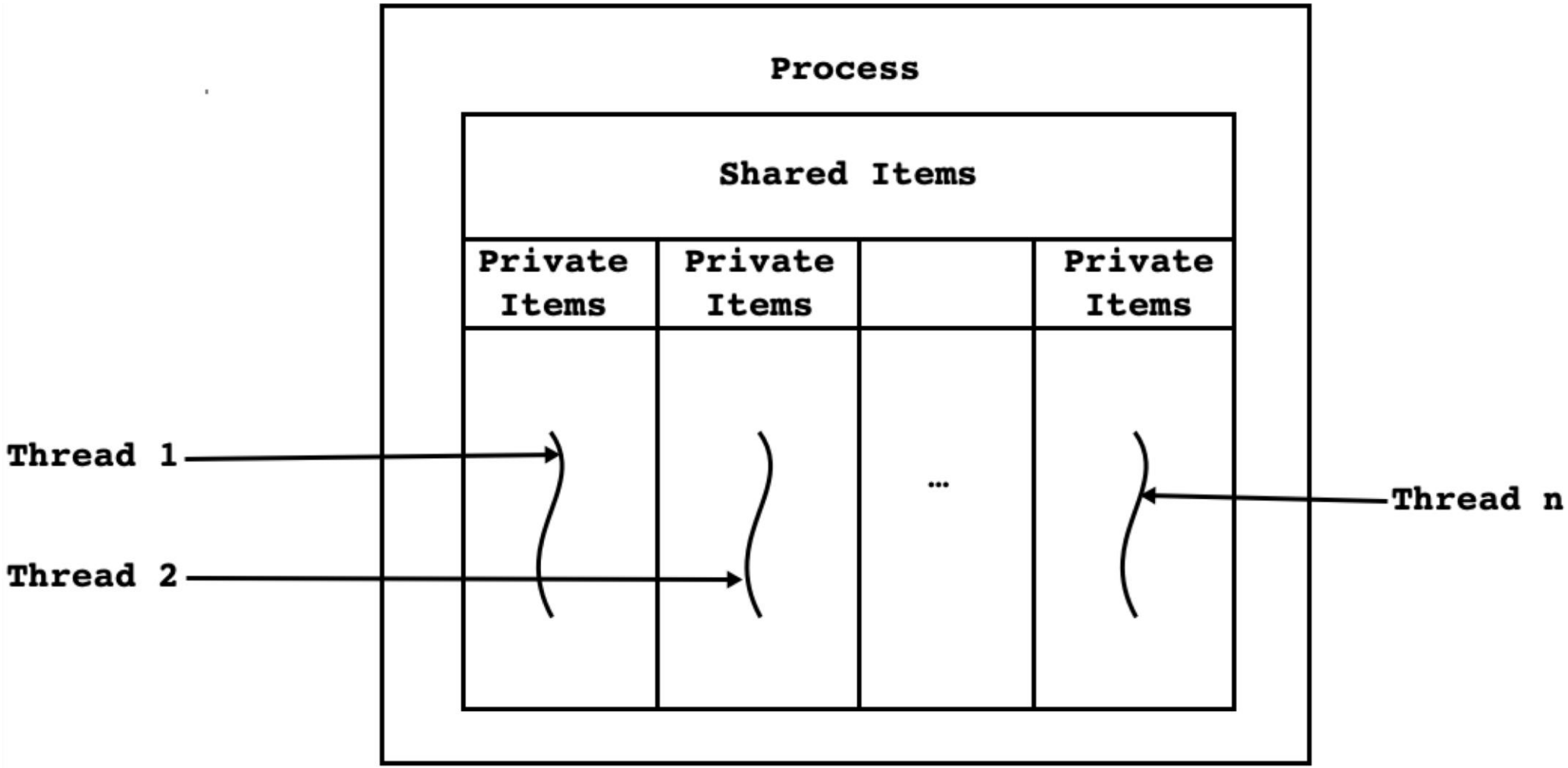
Process Based Multitasking



Process Based Multitasking

- If we want, operating system should execute multiple processes at the same time then OS must first save state of current process into process control block(PCB). Only after that control of CPU can be transferred to the another process. It is called as context switching.
- Creating / disposing process and context switching is a heavy task. Hence process Based multitasking is called heavy weight multitasking.

Thread Based Multitasking



The model of multithreaded Process

Thread Based Multitasking

- In comparison with process, thread creation and disposal/termination take less time.
- To access shared items of process, thread do not require context switching. Hence thread based multitasking is called as light weight multitasking.

Java is *Multithreaded* Programming Language

- When we start execution of Java application, JVM starts execution of main thread and garbage collector.
  - Main thread
    1. It is a *user thread*(also called as non daemon thread).
    2. It is responsible for invoking main method.
  - Garbage Collector(also called as *finalizer*)
    1. It is a *daemon thread*(also called as background thread).
    2. It is responsible for deallocating/releasing/reclaiming memory of unused instances.
- Thread is an OS resource. To access OS thread in application, Java application developer need not to do native programming. Sun/Oracle developers has already developed framework to access OS thread in Java application.

Thread Types

- Java offers two types of thread:
  1. User Thread
  2. Daemon Thread
- User threads are high priority threads. JVM will not terminate until all user threads gets terminated.
- Daemon threads are low priority threads. When thread is marked as daemon, JVM doesn't wait to finish its task.
- Thread inherits properties from its parent. Main thread is user thread hence, if we create any thread from main then it is by default user thread. If we create any thread from daemon thread then it is by default daemon thread.
- Using setDaemon() method we can convert user thread to deamon or vice versa.

Thread Types

- Using isDaemon( ) method, we can check whether thread is user thread or daemon thread.
- We can use setDaemon( ) method to change type of thread. This method should be called after creating thread instance and before calling start method on thread instance.

```
public class Program {
    public static void main(String[] args){
        Thread thread = Thread.currentThread();
        System.out.println(thread.toString());  //Thread[main,5,main]
        if( !thread.isDaemon())
            thread.setDaemon(true); //IllegalThreadStateException
    }
}
```

Thread API

- **Packages**
  1. java.lang
  2. java.util.concurrent
- **Interface**
  - java.lang.Runnable
- **Class(es)**
  - Thread
  - ThreadGroup
  - ThreadLocal
- **Enum**
  - Thread.State
- **Exception**
  - IllegalThreadStateException
  - IllegalMonitorStateException
  - InterruptedException

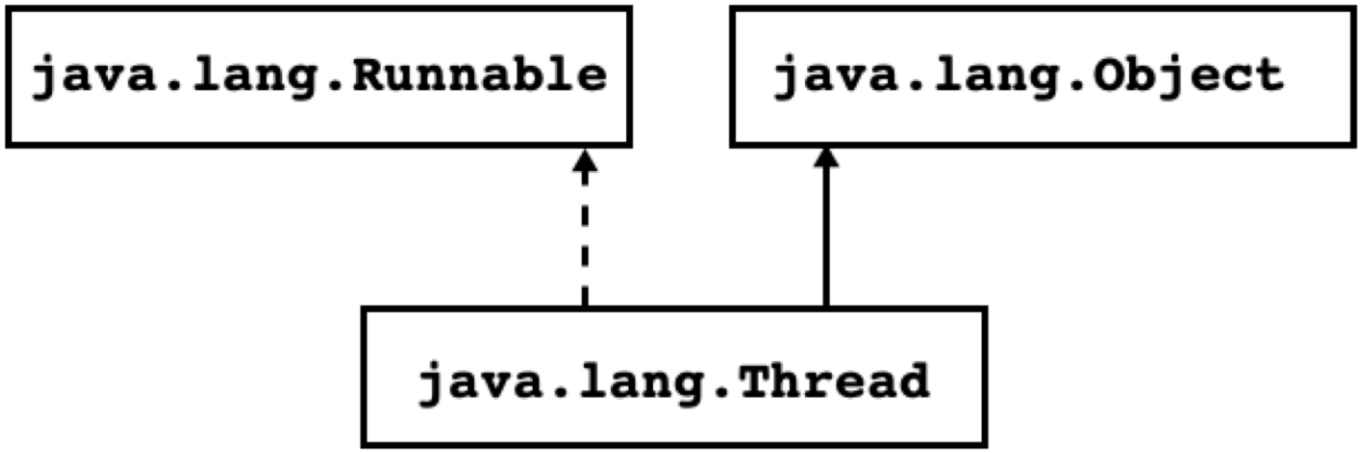
Runnable

- It is a functional interface declared in java.lang package.
- “void run( )” is a method / functional method of Runnable interface.
- Without sub classing java.lang.Thread, if we want to create thread then we should use Runnable interface.

```
class Task implements Runnable{
    private Thread thread;
    public Task( String name ) {
        this.thread = new Thread(this, name);
        this.thread.start();
    }
    @Override
    public void run() {
        //TODO : Write Business Logic Here
    }
}
```

Thread

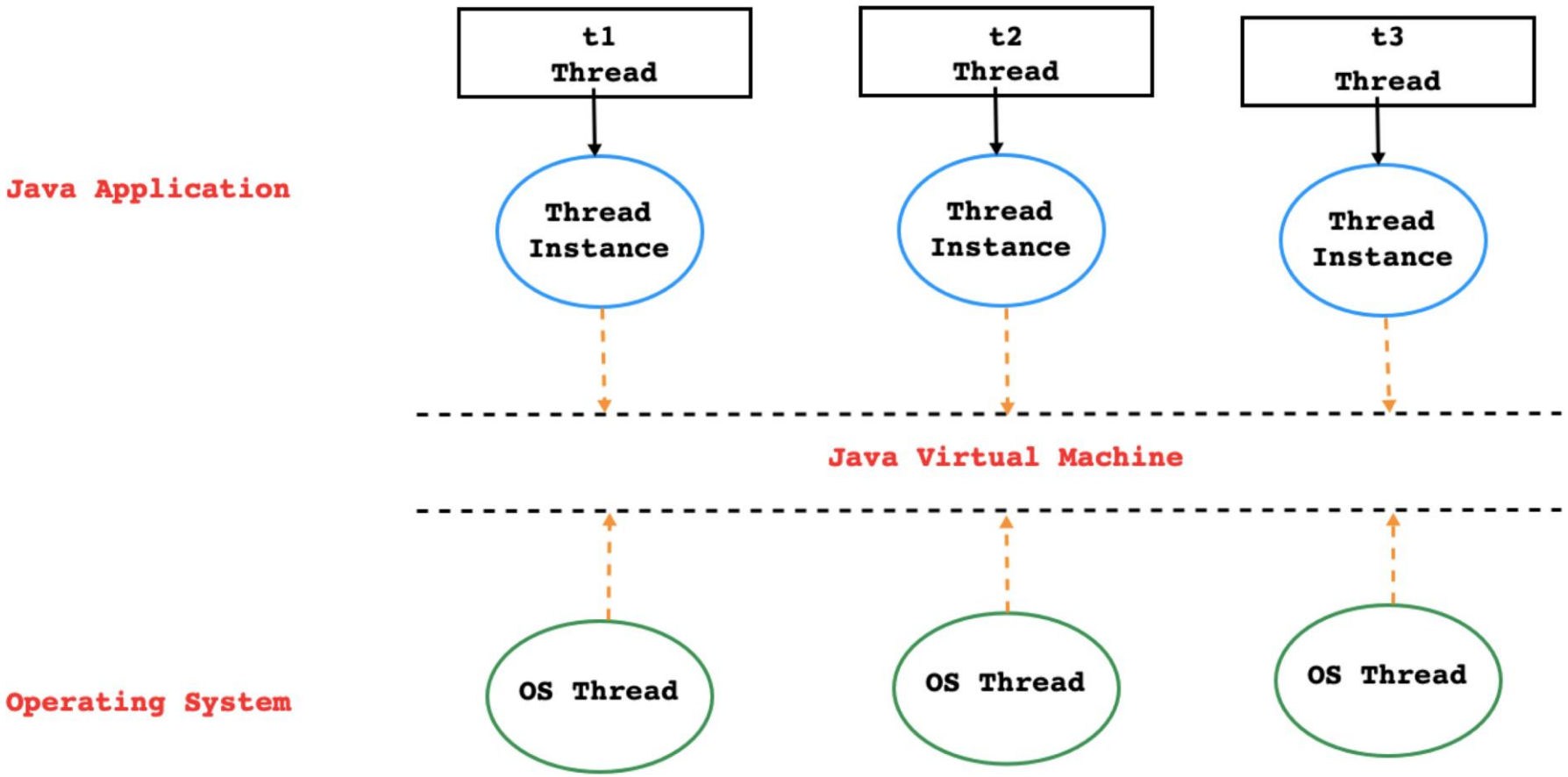
- It is a non final and concrete class declared in java.lang Package.
- In Java language, we can create thread using two ways:
  1. java.lang.Runnable interface
  2. java.lang.Thread class



- Java.lang.Thread is managed version(Java) of unmanaged thread(oS.)

Thread

- Instance of java.lang.Thread class is not an operating System thread. It represents operating System(actually kernel) thread.



Thread

- Thread creation using java.lang.Thread class:

```
class Task extends Thread{
    public Task( String name ) {
        //super( name );    //or
        this.setName(name);
        this.start();
    }
    @Override
    public void run() {
        //TODO : Write Business Logic Here
    }
}
```

Thread.State

- State is a static nested type(Enum) declared inside java.lang.Thread class.
- A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.
  - **NEW**
    - A thread that has not yet started is in this state.
  - **RUNNABLE**
    - A thread executing in the Java virtual machine is in this state.
  - **BLOCKED**
    - A thread that is blocked waiting for a monitor lock is in this state.
  - **WAITING**
    - A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
  - **TIMED_WAITING**
    - A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
  - **TERMINATED**
    - A thread that has exited is in this state.

Fields and Constructor's of Thread Class

- **Field(s)**
  1. public static final int MIN_PRIoRITY
  2. public static final int NoRM_PRIoRITY
  3. public static final int MAX_PRIoRITY

- **Constructor(s)**
  1. public Thread()
  2. public Thread(String name)
  3. public Thread(Runnable target)
  4. public Thread(Runnable target, String name)
  5. public Thread(ThreadGroup group, String name)
  6. public Thread(ThreadGroup group, Runnable target)
  7. public Thread(ThreadGroup group, Runnable target, String name)

Methods Of Thread Class

```
1. public static Thread currentThread()
2. public static void sleep(long millis) throws InterruptedException
3. public void start()
4. public static void yield()
5. public final String getName()
6. public final void setName(String name)
7. public final int getPriority()
8. public final void setPriority(int newPriority)
9.  public Thread.State getState()
10.public static boolean interrupted()
11.public void interrupt()
12.public final boolean isAlive()
13.public final boolean isDaemon()
14.public final void setDaemon(boolean on)
15.public final void join(long millis) throws InterruptedException
16.public void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)
```

Main Thread

```
public class Program {
    public static void main(String[] args){
        Thread thread = Thread.currentThread();
        System.out.println(thread.toString());  //Thread[main,5,main]
        System.out.println("Name     :    "+thread.getName());     //main
        System.out.println("Priority:    "+thread.getPriority());//5
        System.out.println("Group    :    "+thread.getThreadGroup().getName());//main
        System.out.println("State    :    "+thread.getState().name());     //RUNNABLE
        System.out.println("Type     :    "+(thread.isDaemon()?"Daemon":"User"));//User
        System.out.println("Status   :    "+(thread.isAlive()?"Alive":"Dead"));//Alive
    }
}
```

Finalizer / GC

```
class ResourceType{
    @Override
    protected void finalize() throws Throwable {
        Thread thread = Thread.currentThread();
        System.out.println(thread.toString());  //Thread[main,5,main]
        System.out.println("Name     :   "+thread.getName());    //Finalizer
        System.out.println("Priority:   "+thread.getPriority());//8
        System.out.println("Group    :   "+thread.getThreadGroup().getName());//system
        System.out.println("State    :   "+thread.getState().name());    //RUNNABLE
        System.out.println("Type     :   "+(thread.isDaemon()?"Daemon":"User"));//Daemon
        System.out.println("Status   :   "+(thread.isAlive()?"Alive":"Dead"));//Alive
    }
}
public class Program {
    public static void main(String[] args){
        ResourceType rt = new ResourceType();
        rt = null;
        System.gc();
    }
}
```

Thread Creation

Using Runnable Interface

```
class Task implements Runnable{
    private Thread thread;
    public Task( String name ) {
        this.thread = new Thread(this, name);
        this.thread.start();
    }
    @Override
    public void run() {
        //TODO : Write Business Logic Here
    }
}
```

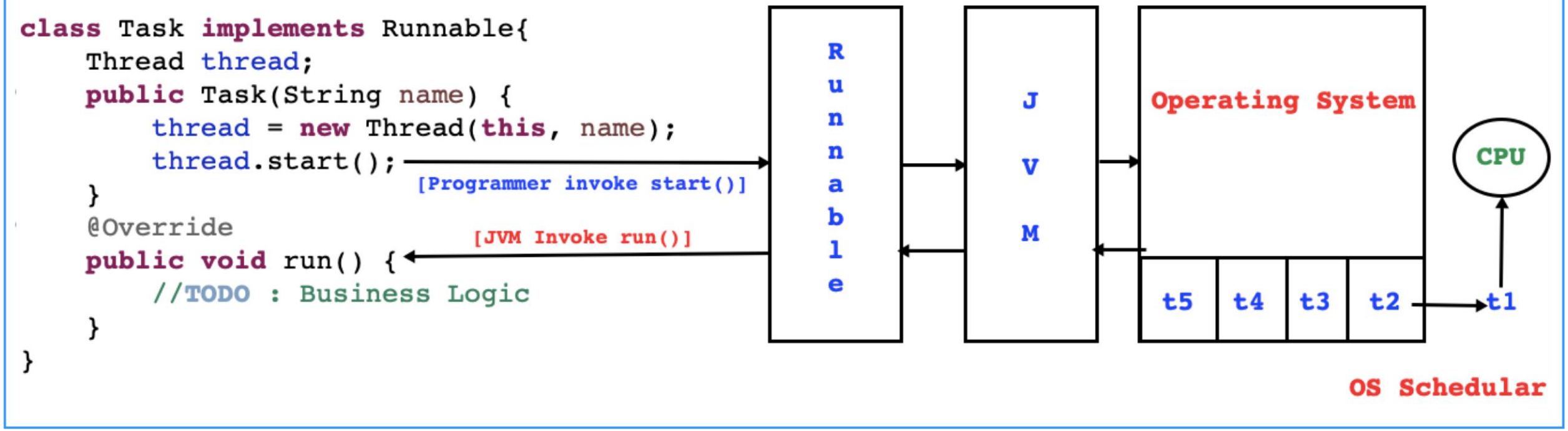
Using Thread class

```
class Task extends Thread{
    public Task( String name ) {
        //super( name );    //or
        this.setName(name);
        this.start();
    }
    @Override
    public void run() {
        //TODO : Write Business Logic Here
    }
}
```

Relation Between start( ) and run( ) method.

- start() method do not call run method.
- If we call start() method on thread instance then JVM ask oS to map java thread instance to oS thread. When oS scheduler assign CPU to oS thread then oS ask JVM to invoke run() method on corresponding java instance.

```
class Task implements Runnable{
    Thread thread;
    public Task(String name) {
        thread = new Thread(this, name);
        thread.start();
    }
    @Override
    public void run() {
        //TODO : Business Logic
    }
}
```



Thread termination

- If control come out of run( ) method then thread gets terminated.

- In following cases thread can come out of run( ) method:
  1. If JVM execute run method successfully.
  2. If JVM throw exception during execution of run method.
  3. If JVM execute return statement during execution of run method.

Blocking calls/Operations

- Following calls are considered as blocking calls in multithreaded programming:
  1. sleep( )
  2. suspend( ) [**Deprecated**]
  3. wait( )
  4. join
  5. input calls
- **sleep( ) method**
  - It is overloaded static method of java.lang.Thread class.
  - Syntax:
    - public static void sleep(long milliSeconds) throws InterruptedException
  - If we want to suspend current thread for specified number of milliseconds then we should use sleep method.
- **suspend( ) method**
  - It is deprecated non static method of java.lang.Thread class.
  - Syntax:
    - public final void suspend( )
  - If we want to suspend any thread for infinite time then we should use suspend(). method.

Blocking calls/Operations

- Following calls are considered as blocking calls in multithreaded programming:
  1. sleep( )
  2. suspend( ) [**Deprecated**]
  3. wait( )
  4. join
  5. input calls
- **sleep( ) method**
  - It is overloaded static method of java.lang.Thread class.
  - Syntax:
    - public static void sleep(long milliSeconds) throws InterruptedException
  - If we want to suspend current thread for specified number of milliseconds then we should use sleep method.
- **suspend( ) method**
  - It is deprecated non static method of java.lang.Thread class.
  - Syntax:
    - public final void suspend( )
  - If we want to suspend any thread for infinite time then we should use suspend(). method.

Interrupting Threads

- "public void interrupt()" is a method of java.lang.Thread class.

- When the interrupt method is called on a thread, the *interrupted status* of the thread is set. This is a boolean flag that is present in every thread.

- "public boolean isInterrupted()" is a method of java.lang.Thread class. It tests whether this thread has been interrupted. The *interrupted status* of the thread is unaffected by this method.

- "public static boolean interrupted()" is a method of java.lang.Thread class. It tests whether the current thread has been interrupted. The *interrupted status* of the thread is cleared by this method. In other words, if this method were to be called twice in succession, the second call would return false.

Interrupting Threads

- However, if a thread is blocked, it cannot check the interrupted status. This is where the InterruptedException comes in.

- When the interrupt method is called on a thread that blocks on a call such as sleep or wait, the blocking call is terminated by an InterruptedException.

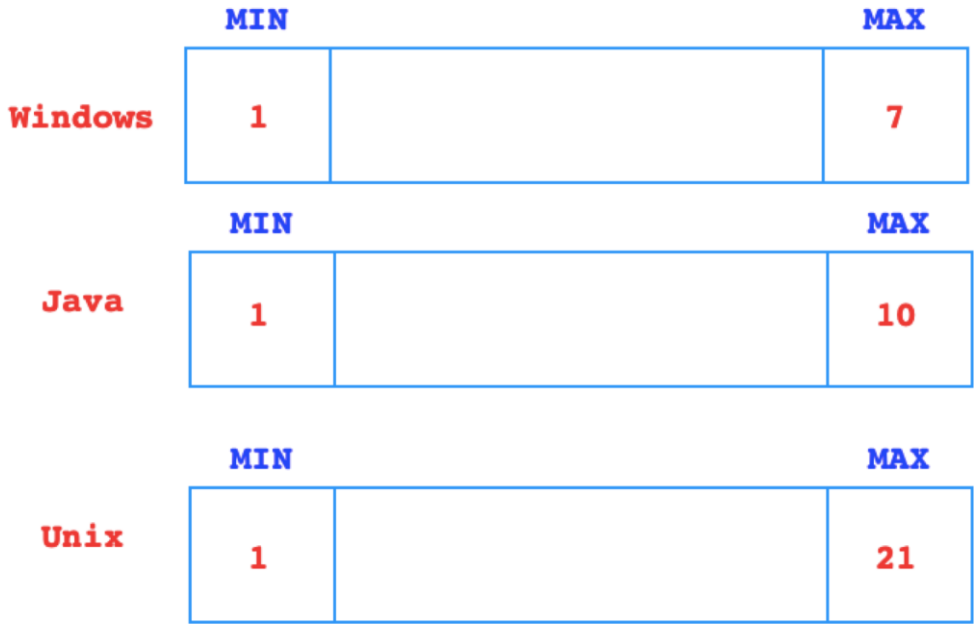
- There is no language requirement that a thread which is interrupted should terminate. Interrupting a thread simply grabs its attention. The interrupted thread can decide how to react to the interruption.

Thread Priorities

- In the Java programming language, every thread has a *priority*.
- Whenever the thread scheduler has a chance to pick a new thread, it prefers threads with higher priority.
- By default, a thread inherits the priority of the thread that constructed it. We can increase or decrease the priority of any thread with the setPriority() method.
- We can set the priority to any value between MIN_PRIoRITY and MAX_PRIoRITY.
- If the priority is not in the range MIN_PRIoRITY to MAX_PRIoRITY the setPriority() method throws **IllegalArgumentException**

Thread Priorities

- Thread priorities are *highly system dependent*.
- When the virtual machine relies on the thread implementation of the host platform, the Java thread priorities are mapped to the priority levels of the host platform, which may have more or fewer thread priority levels.
- In the Oracle JVM for Linux, thread priorities are ignored altogether—all threads have the same priority.



	Java	MS Windows	Unix
t1	8	7	16
t2	7	7	14
t3	2	2	4

Platform Dependent Features

- Following features of the Java makes it platform dependent:
  1. Abstract Window Toolkit(AWT)components
     - AWT components implicitly based on peer classes. These classes are written into C++.
  2. Thread Priorities.
     - JVM thread model is depends on host platform thread model.

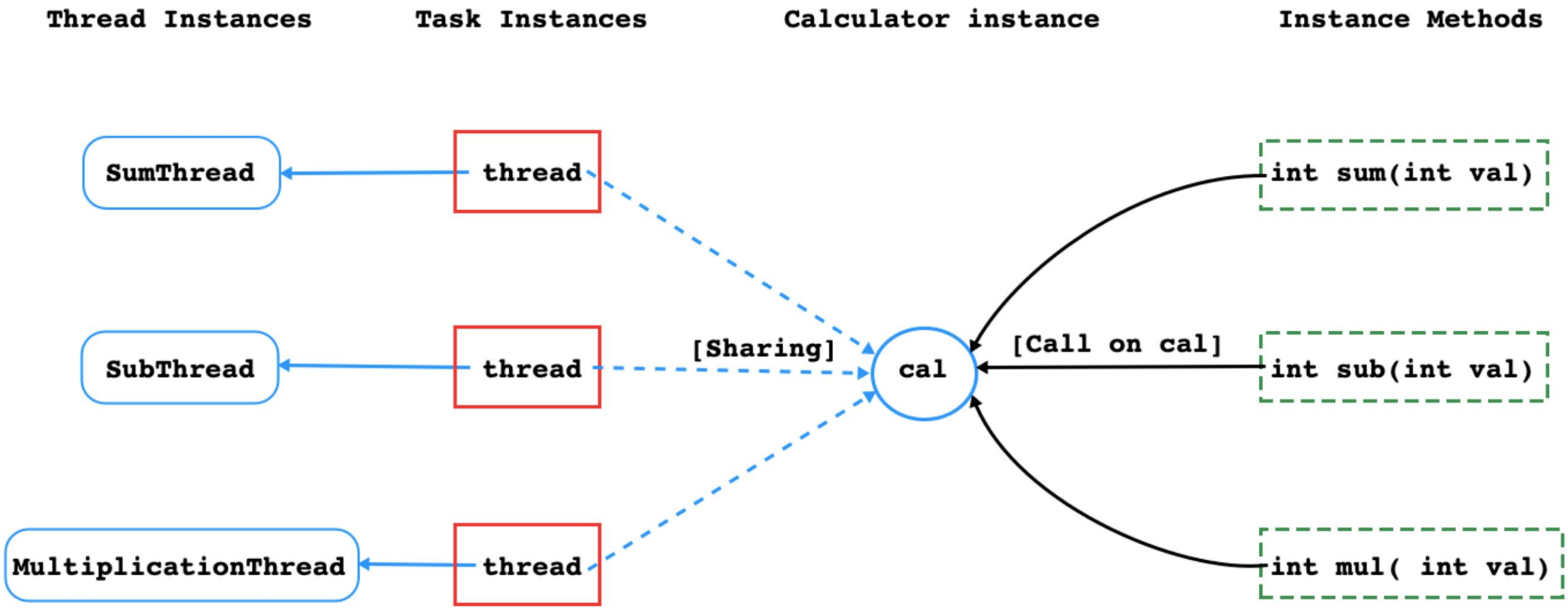
Race Condition

- In most practical multithreaded applications, two or more threads need to share access to the same data. What happens if two threads have access to the same object and each calls a method that modifies the state of the object? As you might imagine, the threads can step on each other's toes.

- Depending on the order in which the data were accessed, corrupted objects can result. Such a situation is often called a *race condition*.

- To avoid corruption of shared data by multiple threads, you must learn how to *synchronize the access*.

Race Condition

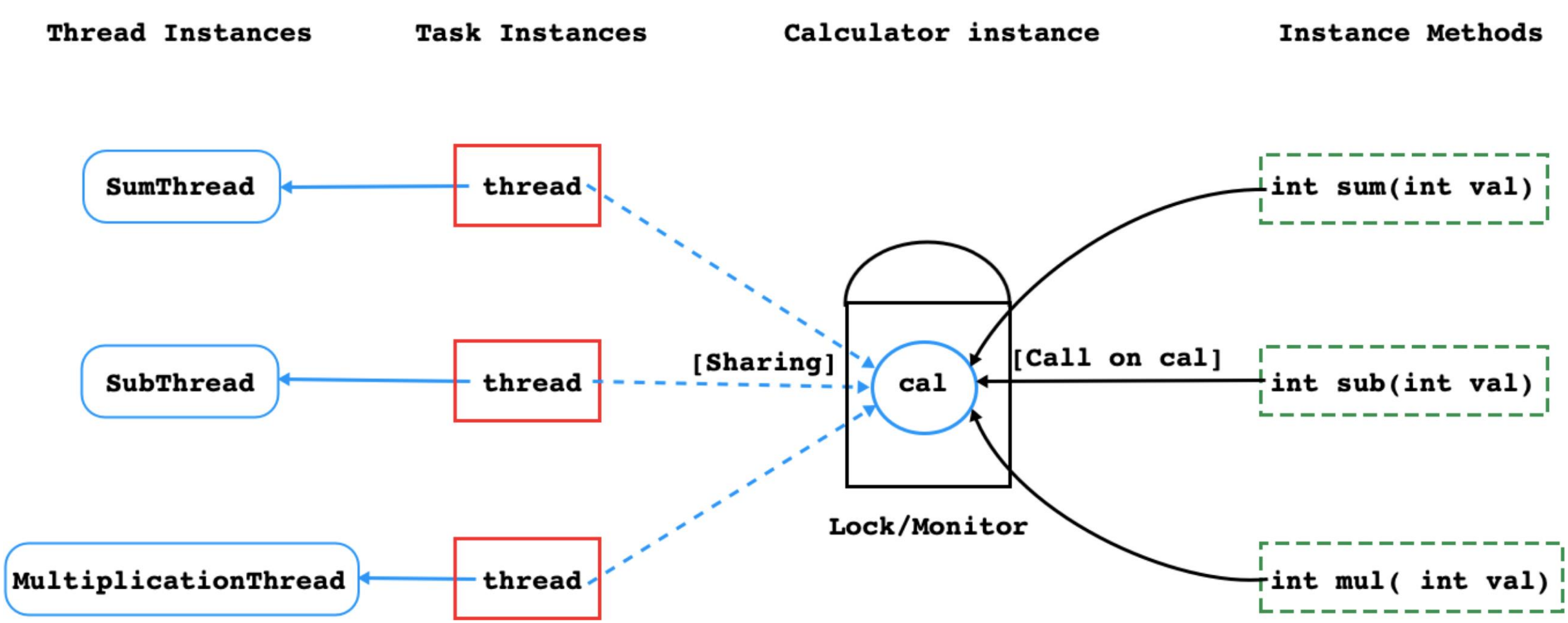


Lock Objects

- There are two mechanisms for protecting a code block from concurrent access.
  1. A synchronized keyword.
  2. The ReentrantLock class.
- The synchronized keyword automatically provides a lock as well as an associated "condition," which makes it powerful and convenient for most cases that require explicit locking.
- The basic outline for protecting a code block with a ReentrantLock is:

```
myLock.lock(); // a ReentrantLock object try
{
    critical section
} finally {
    myLock.unlock(); //lock is unlocked even if an exception is thrown
}
```

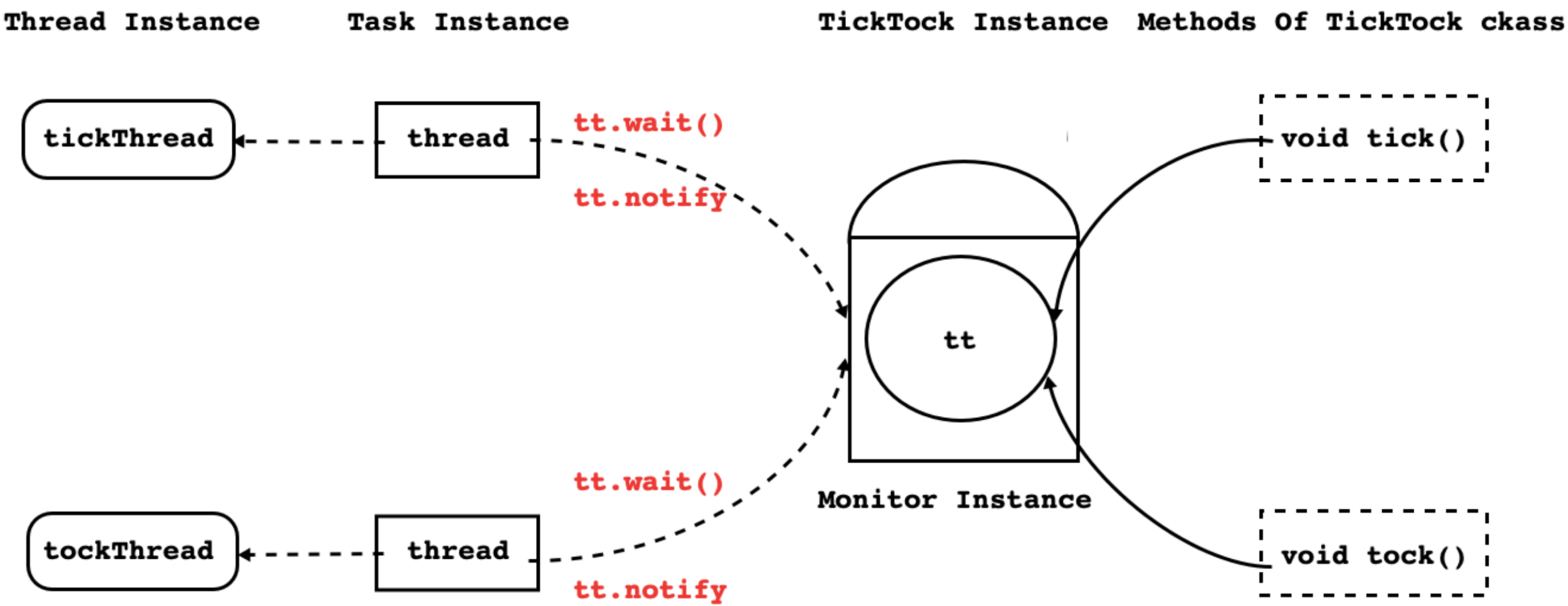
Monitor Concept



Thread Synchronization

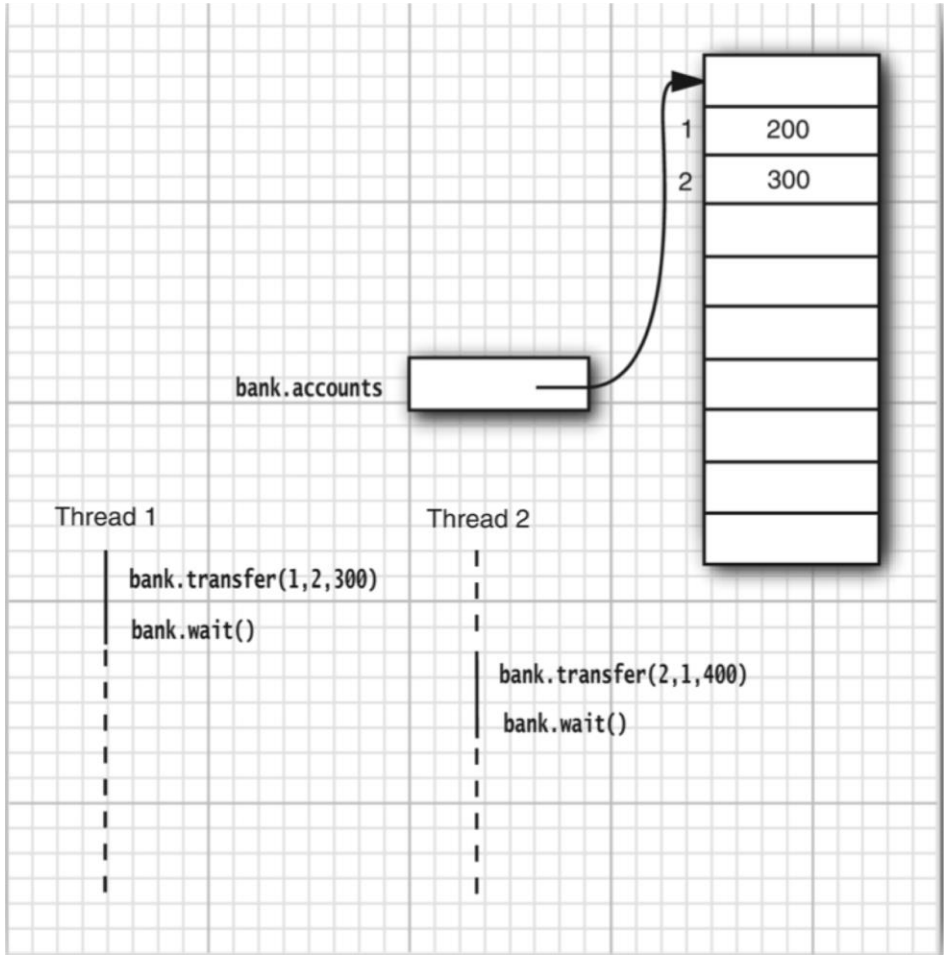
- Using same object's monitor, if multiple threads try to communicate with each other then it is called inter thread communication. And inter thread communication represents synchronization.
- To achieve synchronization, we can use methods declared in java.lang.Object class:
  1. public final void wait() throws InterruptedException
  2. public final void wait(long timeout) throws InterruptedException
  3. public final void wait(long timeout, int nanos) throws InterruptedException
  4. public final void notify()
  5. public final void notifyAll( )
- Resource locking / inter thread communication is based on monitor object associated with shared resource. Type of every shared resource is directly or indirectly extended from java.lang.Object class. Hence wait/notify/notifyAll methods are declared in java.lang.Object class.

Thread Synchronization



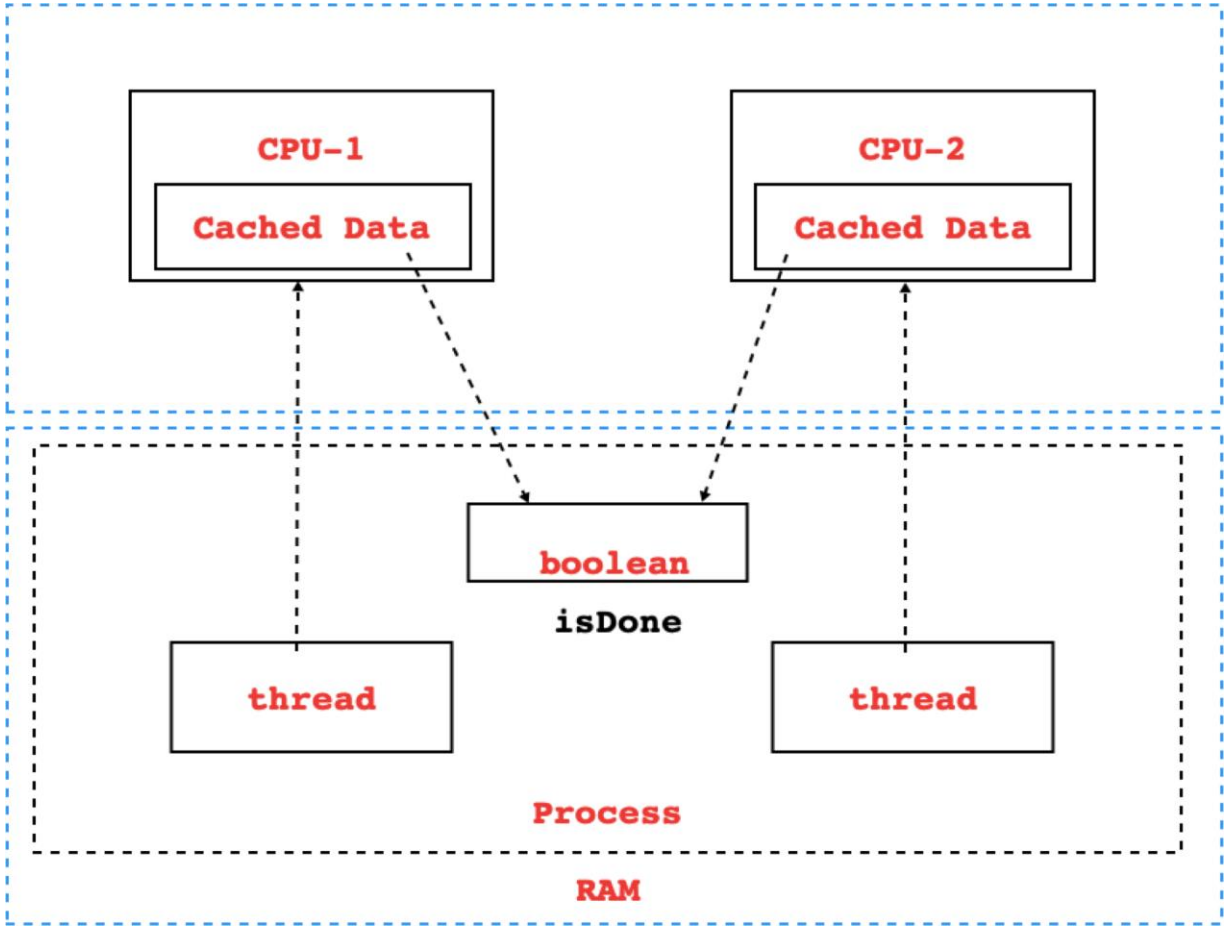
Deadlock

- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other.

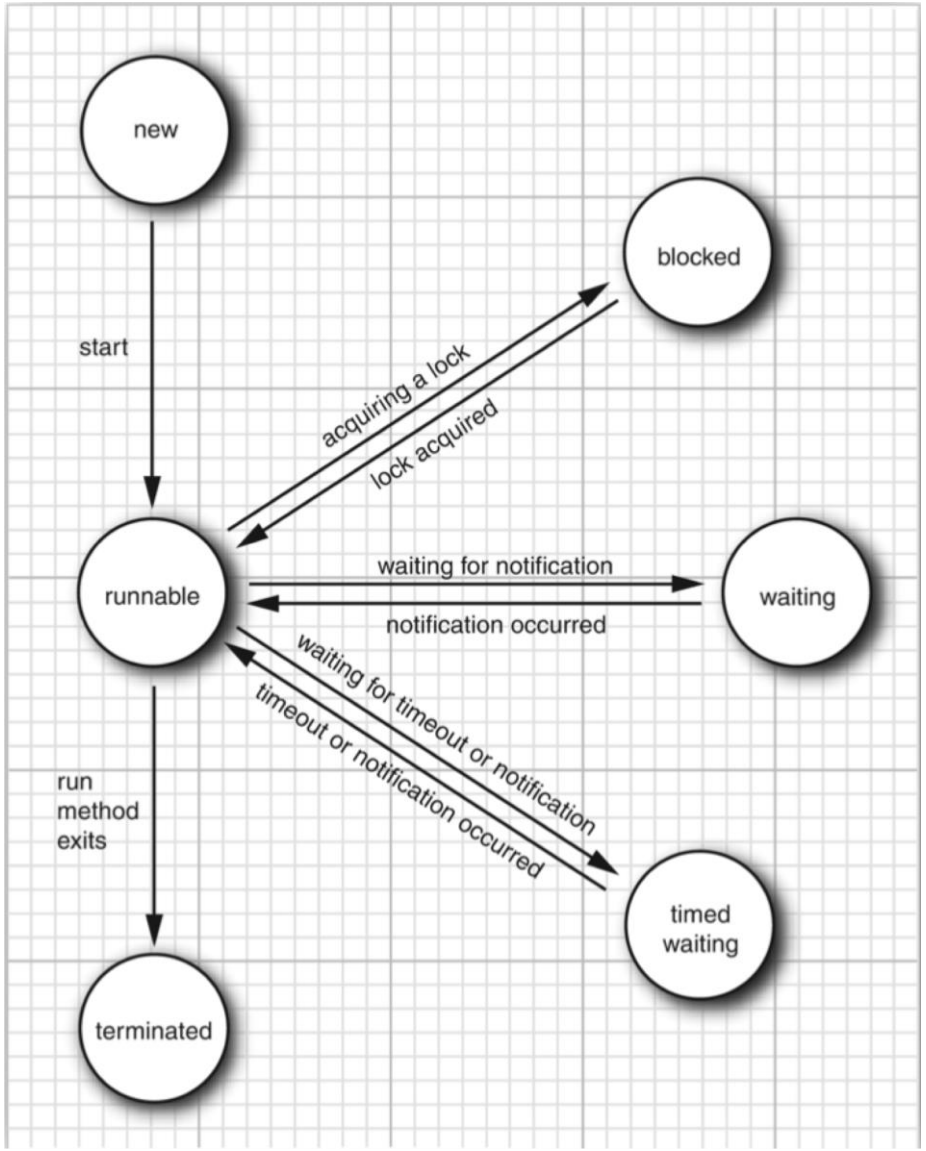


Volatile Fields

- Computers with multiple processors can temporarily hold memory values in registers or local memory caches. As a consequence, threads running in different processors may see different values for the same memory location.



Thread Life Cycle



Thank You.

[akshita.chanchlani@sunbeaminfo.com]